

## A cobbled together LUA Reference Book

Site: [On-Line School of Network Sciences](#)  
Course: Explore Packet Analysis with Wireshark (Standard Edition)  
Book: A cobbled together LUA Reference Book

Printed by: Andrew Walding  
Date: Friday, April 15, 2022, 7:58 AM

## Description

For now, I am collecting various LUA references in this book. There is no strict organization, but having everything here is better than searching in Google.



## Table of contents

1. Introduction
2. An example LUA Dissector
3. Learn LUA in an Hour
4. Three LUA Coding Tutorials
5. LUA Examples and Tutorial from the Wireshark Web Site
6. The main LUA Site and example code pages
7. Wireshark LUA Script to Search for TCP Delays
8. The TRANSUM LUA script for recording various RTT measurements
9. A LUA Script to pull HTTP transaction security and performance data from a packet trace
10. A Simple Tap Script for TShark
11. Thomas Kager's lua Lesson 1: Taping TCP Expert Data
12. Thomas Kager's lua Lesson 2: Filling In the Blanks

# 1. Introduction

If you are a developer of new protocols, or a security expert that wants to push Wireshark into new capabilities, then the embedded LUA interpreter is your answer.

While this course does not teach the interpreter, we mention it. Many students have asked for more information on LUA. This little reference hopes to answer that request.



## What is LUA?

Lua is a powerful, fast, lightweight, embeddable scripting language. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and [rapid](#) prototyping.

## Where does Lua come from?

Lua is designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. Lua was born and raised in Tecgraf, formerly the Computer Graphics Technology Group of PUC-Rio. Lua is now housed at LabLua, a laboratory of the Department of Computer Science of PUC-Rio.

## What's in a name?

"Lua" (pronounced LOO-ah) means "Moon" in Portuguese. As such, it is neither an acronym nor an abbreviation, but a noun. More specifically, "Lua" is a name, the name of the Earth's moon and the name of the language. Like most names, it should be written in lower case with an initial capital, that is, "Lua". Please do not write it as "LUA", which is both ugly and confusing, because then it becomes an acronym with different meanings for different people. So, please, write "Lua" right!

You will note that this information is organized in chapters, but that is not to say that a [lot](#) of thought (at this point anyway) has gone into it's organization.

If you can think of additional information that should go into this reference, please let us know.

## 2. An example LUA Dissector

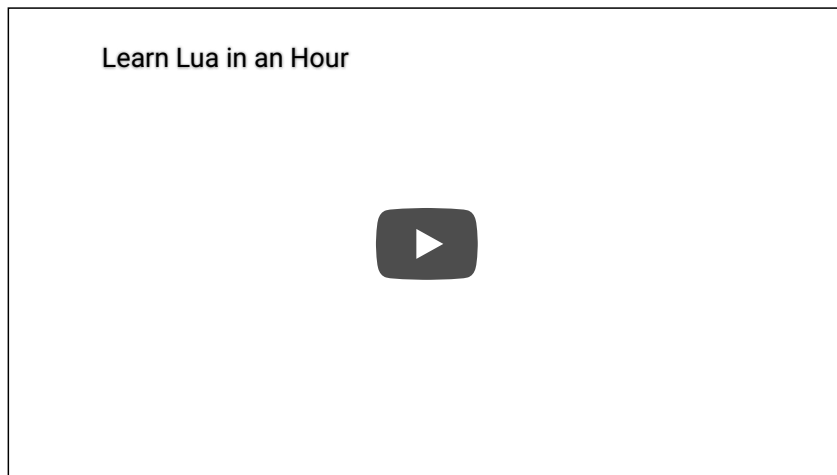
In this video, you will see an example LUA dissector implementation. It will help you get an idea of what LUA is about.

**Packet Class: Wireshark - Lua Protocol Dissectors**



### 3. Learn LUA in an Hour

Hopefully this video will help you begin to learn LUA:



## 4. Three LUA Coding Tutorials

Watch these in order:

**Lua Coding Tutorial 01 - Setting up your workspace**



**Lua Coding Tutorial 02 - Variables and Functions**



**Lua Coding Tutorial 03 - Loops**



## 5. LUA Examples and Tutorial from the Wireshark Web Site

Start with this page: [LUA Intro from the Wireshark Web Site](#)

Then go to the following link: [Lua Examples and Tutorial from the Wireshark Web Site](#)

Here is the [link to the LUA API at Wireshark](#).



## 6. The main LUA Site and example code pages

You can find the [main Lua web page here](#).

You can find the [example LUA code pages here](#).

You can [download LUA here](#).

You can access the LUA [manuals online here](#).

## 7. Wireshark LUA Script to Search for TCP Delays

The following is lifted from the following web

page: [https://www.ibm.com/developerworks/community/blogs/kevgrig/entry/wireshark\\_lua\\_script\\_to\\_search\\_for\\_tcp\\_delays?lang=en](https://www.ibm.com/developerworks/community/blogs/kevgrig/entry/wireshark_lua_script_to_search_for_tcp_delays?lang=en)

Credit and Authorship goes to: Kevin Grigorenko

I've wanted to write this for a long time and I finally had a situation that called for it. The following tshark Lua script searches network packet captures for anomalous [TCP](#) delays in handshakes (long response time to a SYN, response not a SYN/[ACK](#), missing response to a SYN, duplicate SYN) and delays between packets after a handshake. The latter is disabled by default because connections are often [re-used](#), so there may be legitimate delays between the end of one part of a conversation and the beginning of another, so that requires more delicate analysis.

```
-- Usage: tshark -q -r ${CAPTURE_FILE} -X lua_script:tcpanomalies.lua
--
-- tcpanomalies.lua: Find all TCP tcpanomalies and:
-- 1) Print the time it takes for a response to a SYN (either direction). Modify
minSynResponseDelta in script to only print those longer than X seconds,
-- 2) Print an ERROR if the response to a SYN is not SYNACK (e.g. RST),
-- 3) Print an ERROR if a SYN is sent on the same stream without a response to a previous SYN,
-- 4) Print a WARNING if a SYN does not receive a response by the end of the capture (only warning
because capture may have ended right before a legitimate response).
-- Notes:
-- 1) We look at each 4-tuple of (source IP, source port, destination IP, destination port) -
Wireshark calls this a "stream" and conveniently numbers each tuple uniquely for us (tcp.stream),
-- 2) By default, we suppress the warning if the script finds packets on a stream without a
previous SYN as these are probably at the start of the capture for streams we didn't capture
tcpanomalies for.
-- 3) Only works on a single file, so use mergecap to merge rolling files together.
--
-- Example:
-- $ tshark -q -r http1-rstnoack.pcap -X lua_script:tcpanomalies.lua
-- tcpanomalies.lua: Started
-- tcpanomalies.lua: First packet time: "Mar 22, 2014 07:08:36.967090000 PDT"
-- tcpanomalies.lua: =====
-- tcpanomalies.lua: ERROR: Received RST in response to SYN after 2.4080276489258e-05 seconds.
First SYN sent: "Mar 22, 2014 07:08:36.967090000 PDT", Stream 0, Frame: 1, Source: nil:36016,
Destination: nil:80, Current frame time: "Mar 22, 2014 07:08:36.967114000 PDT" (Frame: 2)
-- tcpanomalies.lua: WARNING: Stream 1 did not get a response by the end of the capture. First SYN
sent: "Mar 22, 2014 07:08:36.967251000 PDT", Stream 1, Frame: 3, Source: 127.0.0.1:45317,
Destination: 127.0.0.1:80, Current frame time: (Frame: )
-- tcpanomalies.lua: =====
-- tcpanomalies.lua: Last packet time: "Mar 22, 2014 07:08:36.967251000 PDT"
-- tcpanomalies.lua: Finished

local suppressMissingHandshake = true

-- Update this to search for long SYN response times. In seconds, e.g. .0000250
local minSynResponseDelta = 1

-- Update this to search for gaps between packets after the handshake. In seconds, e.g. .0000250
local minDiffDelta = 0

-- Internal variable
local lastTime = nil

do
function scriptprint(message)
print("tcpanomalies.lua: " .. message)
end

scriptprint("Started tcpanomalies.lua")

-- frame
local frame_time = Field.new("frame.time")
local frame_number = Field.new("frame.number")
local frame_len = Field.new("frame.len")
local frame_epochtime = Field.new("frame.time_epoch")
```

```

-- tcp
local tcp_dstport = Field.new("tcp.dstport")
local tcp_srcport = Field.new("tcp.srcport")
local tcp_stream = Field.new("tcp.stream")
local tcp_pdu_size = Field.new("tcp.pdu.size")
local tcp_flags_syn = Field.new("tcp.flags.syn")
local tcp_flags_ack = Field.new("tcp.flags.ack")
local tcp_flags_rst = Field.new("tcp.flags.reset")
local tcp_flags_fin = Field.new("tcp.flags.fin")

-- ipv4
local ip_dst = Field.new("ip.dst")
local ip_src = Field.new("ip.src")

local streams = {}

local function init_listener()
local tap = Listener.new("tcp")
function tap.reset()
-- print("tap reset")
end

function tap.packet(pinfo,tvb)
local dstport = tcp_dstport()
local srcport = tcp_srcport()
local frametime = tostring(frame_time())
local frameepochtime = tonumber(tostring(frame_epochtime()))
local framenumber = tonumber(tostring(frame_number()))
local ipdst = ip_dst()
local ipsrc = ip_src()
local stream = tonumber(tostring(tcp_stream()))
local flagssyn = tonumber(tostring(tcp_flags_syn()))
local flagsack = tonumber(tostring(tcp_flags_ack()))
local flagsrst = tonumber(tostring(tcp_flags_rst()))
local flagsfin = tonumber(tostring(tcp_flags_fin()))

if lastTime == nil then
scriptprint("First packet time: " .. frametime)
scriptprint("=====")
end
lastTime = frametime

-- First check if this is a new connection: SYN and not ACK
if flagssyn == 1 and flagsack == 0 then

-- Check if there's any previous machine state
if machine ~= nil then
if machine.state == 1 then
scripterror(stream, machine, "Never received SYN response before a new SYN", frametime,
framenumber)
end
end

streams[stream] = {
frame = framenumber,
time = frametime,
source = tostring(ipsrc) .. ":" .. tostring(srcport),
destination = tostring(ipdst) .. ":" .. tostring(dstport),
start = frameepochtime,
state = 1,
lastpackettime = frameepochtime,
lastpacketframe = framenumber,
lastpacketdatetime = frametime
};
else
-- Otherwise, use the state machine

local machine = streams[stream]

```

```

if machine ~= nil then
if machine.state == 1 then
-- Only have a SYN so far, so we expect this to be a SYN ACK
local diff = frameepochtime - machine.start
if flagsack == 1 and flagssyn == 1 then
if diff >= minSynResponseDelta then
postsyn(stream, machine, "SYNACK", frametime, framenumber, diff)
end
machine.state = 2
elseif flagsrst == 1 then
if diff >= minSynResponseDelta then
scripterror(stream, machine, "Received RST in response to SYN after " .. diff .. " seconds",
frametime, framenumber)
end
machine.state = 2
elseif flagsfin == 1 then
scriptwarning(stream, machine, "Received FIN in response to SYN after " .. diff .. " seconds",
frametime, framenumber)
machine.state = 2
else
scripterror(stream, machine, "Expected SYNACK or RST, instead got frame " .. framenumber .. "
after " .. diff .. " seconds", frametime, framenumber)
machine.state = 3
end
elseif machine.state == 3 then
-- This state means that we've already reported an error on this stream, so we only report the
first error
else
-- Check for delta between any other two packets
local diff = frameepochtime - machine.lastpackettime

if minDiffDelta > 0 and diff >= minDiffDelta then
local passMachine = machine
if machine.state == -1 then
passMachine = nil
end
scriptwarning(stream, passMachine, "Time between frames " .. machine.lastpacketframe .. " (" ..
machine.lastpacketdatetime .. ") and " .. framenumber .. " (" .. frametime .. ") is " .. diff .. "
seconds.", frametime, framenumber)
end

machine.lastpackettime = frameepochtime
machine.lastpacketframe = framenumber
machine.lastpacketdatetime = frametime
end
else
-- We haven't seen a handshake on this stream, but we track it anyway to find packet diffs
streams[stream] = {
state = -1,
lastpackettime = frameepochtime,
lastpacketframe = framenumber,
lastpacketdatetime = frametime
};
if not suppressMissingHandshake then
scriptwarning(stream, nil, "Frame " .. framenumber .. " did not have matching SYN", frametime,
framenumber)
end
end
end

function postsyn(stream, machine, response, responsetime, responseframe, diff)
scriptprint("Stream " .. stream .. " received SYN response (" .. response .. ") after " .. diff ..
" seconds. SYN sent: " .. machine.time .. " (Frame: " .. machine.frame .. "), Source: " ..
machine.source .. ", Destination: " .. machine.destination .. ", Response time: " .. responsetime
.. " (Frame: " .. responseframe .. ")")
end

```

```
function scripterror(stream, machine, message, curtime, curframe)
scriptalert("ERROR", stream, machine, message, curtime, curframe)
end

function scriptwarning(stream, machine, message, curtime, curframe)
scriptalert("WARNING", stream, machine, message, curtime, curframe)
end

function scriptalert(alert, stream, machine, message, curtime, curframe)
if machine ~= nil then
if curtime ~= nil then
scriptprint(alert .. ": " .. message .. ". First SYN sent: " .. machine.time .. ", Stream " ..
stream .. ", Frame: " .. machine.frame .. ", Source: " .. machine.source .. ", Destination: " ..
machine.destination .. ", Current frame time: " .. curtime .. " (Frame: " .. curframe .. ")")
else
scriptprint(alert .. ": " .. message .. ". First SYN sent: " .. machine.time .. ", Stream " ..
stream .. ", Frame: " .. machine.frame .. ", Source: " .. machine.source .. ", Destination: " ..
machine.destination)
end
else
if curtime ~= nil then
scriptprint(alert .. ": " .. message .. ". Stream " .. stream .. ", Current frame time: " ..
curtime .. " (Frame: " .. curframe .. ")")
else
scriptprint(alert .. ": " .. message .. ". Stream " .. stream)
end
end
end

function tap.draw()
-- Check for any SYNs without a response
for stream,machine in pairs(streams) do
if machine.state == 1 then
scriptwarning(stream, machine, "Stream " .. stream .. " did not get a response by the end of the
capture", nil, nil)
end
end

if lastTime ~= nil then
scriptprint("=====")
scriptprint("Last packet time: " .. lastTime)
end

scriptprint("Finished")
end
end

init_listener()
end
```

## 8. The TRANSUM LUA script for recording various RTT measurements

TRANSUM is a LUA script that you can get from <http://www.tribelab.com/transum>

The example on the tribelab webpage does a good job explaining the measurement terminology.

When a client process, such as Internet Explorer, sends an application request message, say an [HTTP](#) POST to a web server, there are four elements of the overall response time:

- **APDU Response Time** - the total time the client must wait for completion of the request
- **Service Time** - the time it takes for the service to process the request
- **Request Spread** - the time needed to transport the whole request APDU from the client to the service
- **Response Spread** - the time needed to transport the whole response APDU from the service to the client

As of today, the script supports the following protocols:

- Web [HTTP](#) and [HTTPS](#)
- Web service [HTTP](#) and [HTTPS](#)
- Microsoft SQL database TDS without Multiple Active Result Set (MARS) activated
- Oracle database TNS
- PostgreSQL database
- .NET Remoting both SOAP and binary
- [SMTP](#)
- [FTP](#) command connection
- Many proprietary protocols that obey the flip-flop pattern

The way to use the TRANSUM LUA script is to launch Wireshark (or even better create a shortcut) like:

```
"C:\Program Files\Wireshark\Wireshark.exe" -X lua_script:transum.lua
```

## 9. A LUA Script to pull HTTP transaction security and performance data from a packet trace

```
--[[
HTTP Expert 1.1
Written by Thomas Kager
tkager@linux.com

Created 3/11/2016
Last modified 3/12/16

- Purpose
The purpose of this script is to pull HTTP transaction security and performance data from a packet trace.

- Usage
tshark -r "(filename)" -2 -X lua_script:http_expert.lua -q > (filename.csv)

    filename = capture file
    filename.csv = destination csv. The output of this script is CSV for easy import into spreadsheet
program such as Excel. While it is possible to output to terminal,
    file redirection (>) is encouraged due to variable field length and readability concerns.

- Requirements
Requires Wireshark/Tshark 1.10 or later with LUA compiled. This can be determined through the "About Wireshark"
menu option.
--]]

tap=Listener.new(nil, "http") -- Create Listener, with a Filter for HTTP traffic (only).

http = {} -- Create usrdta array http.

-- Field Extractors are used to map tshark fields to variables and typically behave as function calls

-- Information collected/derived from HTTP request
http_request_time=Field.new("frame.time")
http_request_frame=Field.new("frame.number")
client=Field.new("ip.src")
server=Field.new("ip.dst")
http_request_method=Field.new("http.request.method")
http_request_version=Field.new("http.request.version")
http_user_agent=Field.new("http.user_agent")
http_host=Field.new("http.host")
http_request_uri=Field.new("http.request.uri")
http_data=Field.new("http")
http_request_header_fields=0 -- As this is a caculated field, there is no direct mapping to tshark field
extractor. We need to create it but we will just initialize it to 0.

-- Information collected from HTTP response
http_reply_frame=Field.new("frame.number")
http_request_in=Field.new("http.request_in")
http_response_code=Field.new("http.response.code")
http_cache_control=Field.new("http.cache_control")
http_time=Field.new("http.time")

function tap.draw() -- Wireshark/Tshark explicitly looks for tap.draw() after running through all packets.
-- Header Output. Needs to occur once before we iterate through the array(s) in the main loop.
io.write("request frame"," ","request time"," ","client"," ","server"," ","request method"," ","request
version"," ","http host"," ","request uri"," ","user agent"," ","request header fields"," ","response
frame"," ","response code"," ","response time"," ","cache control")
io.write("\n") --- linespace after header. This can occur within previous write operation.

-- Main Loop
for k,v in pairs (http) do
-- Optimal to combine these into a single IO write. Such a write can be extended across multiple lines, however
this convention breaks prior to LUA 5.2.
io.write(tostring(k)," ",tostring(http[k][http_request_time])," ",tostring(http[k]
[client])," ",tostring(http[k][server])," ",tostring(http[k][http_request_method])," ",tostring(http[k]
[http_request_version])," ",tostring(http[k][http_host])," ",tostring(http[k]
[http_request_uri])," ",tostring(http[k][http_user_agent])," ",tostring(http[k]
[http_request_header_fields])," ")
io.write(tostring(http[k][http_reply_frame])," ",tostring(http[k][http_response_code])," ",tostring(http[k]
[http_time])," ",tostring(http[k][http_cache_control]))
io.write("\n") --- linespace after row. This can also occur as part of one large write operation.
end

end -- end tap.draw()

function tap.packet() -- Wireshark/Tshark explicitly looks for tap.packet(). It runs for each frame that
matches listener filter.

if http_request_method() then -- If frame is an HTTP request, there are specific fields that we need to
collect.
```



```

request_frame=tostring(http_request_frame())
http[request_frame]={}
http[request_frame][http_request_time]=tostring(http_request_time()):gsub(',','')
http[request_frame][client]=tostring(client())
http[request_frame][server]=tostring(server())
http[request_frame][http_request_method]=tostring(http_request_method())
http[request_frame][http_request_version]=tostring(http_request_version())
http[request_frame][http_host]=tostring(http_host())
http[request_frame][http_request_uri]=tostring(http_request_uri())

-- Determine Number of Request Header Fields.
x=tostring(http_data())
_, count = string.gsub(x, "\0d:\0a", " ") -- Count number of CR/LF, as these delineate header fields.
_, double_white = string.gsub(x, "\0d:\0a:\0d:\0a", " ") -- Count occurrences in which 2 CR/LF occur one
after these other, as these will be counted as 2 header fields.
http[request_frame][http_request_header_fields]=count - double_white - 1 -- Subtract multiple CR/LF
occurrences from the CR/LF count. Also subtract 1, because there is an occurrence between (method, URI, version)
and the first header.

-- Add user_agent if present within headers store the value, else populate with none. This is necessary
as we will get an error if the header field doesn't exist.
if http_user_agent() == nil then
    http[request_frame][http_user_agent]="none"
else
    http[request_frame][http_user_agent]=tostring(http_user_agent())
end

else if http_response_code() then -- If frame is an HTTP response, there are specific fields which we need to
collect.
request_in=tostring(http_request_in())
http[request_in][http_reply_frame]=tostring(http_reply_frame())
http[request_in][http_response_code]=tostring(http_response_code())
http[request_in][http_time]=tostring(http_time())
-- Check for cache control. If it doesn't exist, store none. Else store the value.
if http_cache_control() == nil then
    http[request_in][http_cache_control]="none"
else
    http[request_in][http_cache_control]=tostring(http_cache_control()):gsub(',','') --- We need to
strip out any commas that may exist in cache control header, as this is a CSV.
end

else -- Other frames (such as continuation frames) do not usable field values. We will break the script if we
try and process them.
end

end

end -- end of tap_packet()

```

## 10. A Simple Tap Script for TShark

The following link is to a blog page that contains the explanation behind a simple Tap script. The script, called streamXpert1.lua, examines each [TCP](#) segment, keeps a count of the number of total frames, retransmissions, duplicate acks, out of orders and zero windows by connection (stream).

[You can view the blog here!](#)

[You can download the Lua script here.](#)

# 11. Thomas Kager's lua Lesson 1: Taping TCP Expert Data

## lua Lesson 1 – Tapping TCP Expert data

Kager

by Thomas P. Kager

In this blog, I am going to introduce Lua tap scripting for Tshark. Specifically, this blog is intended to provide a conceptual overview and foundation for more complex development tasks, which will be presented in future blogs.

[Download LUA script](#)

### Introduction

Wireshark is a great tool for analyzing packet captures. However, there are many cases in which Wireshark doesn't represent the data in a usable and efficient manner for the given task at hand. Thus, I have often found myself exporting the data to text, using a language such as Python or Perl to perform string parsing and/or pulling the data into Excel for processing and correlation. More recently I have been using Lua for these sorts of tasks.

[LUA](#) is an embedded, interpreted language which is frequently used in the gaming world. While it bears many similarities to languages such as Python, it has fewer data types and built in functions, etc. The sparse nature of Lua keeps it very small, stable and efficient. Coming from more of a Python and C background, I experienced a period of adjustment, though the basic concepts of programming, e.g. control structures, operators, functions, string parsing, etc. apply. The Lua language/interpreter has direct integration with Wireshark. While this integration allows for the creation of dissectors, we are going to focus on taps as we are processing data that has already been dissected/decoded by Wireshark/Tshark. The [Wireshark Wiki](#) provides additional information on the integration of LUA.

To determine whether or not your version of Wireshark is compiled with Lua, examine "About Wireshark" under help. Here is my about page, with the pertinent text highlighted:



I mentioned the terms dissector and tap, as if this all makes sense to everyone reading this blog. As I realize that this may not be the case, let me take a step back and discuss these components.

Wireshark/Tshark has a core engine which processes each frame, one frame at a time. For each frame Wireshark/Tshark invokes dissectors, plugins, filters and taps.

- dissectors decode the various fields and their values
- plugins are typically external dissectors (dissectors which are not part of core distribution)
- filters restrict the data displayed
- taps work with dissected data, often to create statistics.

When a frame is processed, dissectors and plugins populate an ephemeral "protocol tree" with the various decoded values from the current frame. In a typical scenario, the frame dissector is called which populates the tree with frame related data, such as the arrival time, capture length, etc. If the frame type is Ethernet, the Ethernet dissector is called. If the type is [IP](#), the [IP](#) dissector, called, etc. This occurs until there are no further dissectors available and/or frame data left for dissection. Filters and taps can retrieve information from the tree, but do not have the ability of updating the tree. Filters are used to limit which frames are displayed/tapped and taps are often used to create additional metadata from dissected data.

The following graphic summarizes this relationship:



### Lua Taps


From the command line, we can run a Lua tap script as follows:

```
tshark -X lua_script:streamXpert1.lua -r "small.pcapng" -q
```

This will cause Tshark to initialize the Lua script "streamXpert.lua" and process the trace file named "small.pcapng". The "q" option prevents Tshark from displaying frames to the console, therefore only printing the output of our script. For each frame, Tshark will look for the .packet() function contained in the Lua script. Tap.packet() "extracts" field data from each packet, stores this data, and optionally performs calculations, etc. After all packets have been processed, Tshark looks for a .draw() function for output. Thus, the components of a useful Tshark Lua script may likely include the following :

- Tap definition
- Field extractors
- array to hold extracted data and potentially other variables for program flow control, etc.
- .packet() function
- .draw() function

#### streamXpert1.lua

streamXpert1.lua examines each [TCP](#) segment, keeps a count of the number of total frames, retransmissions, duplicate acks, out of orders and zero windows by connection (stream). Our output should look similar to this: 

### Initialization

At the beginning of our script, we define a tap using [listener.new\(\)](#). Specifically, we are going to be tapping [tcp](#) traffic and our tap is going to be named "tap". Subsequently, if we named our tap "my\_tap", our functions would be named "my\_tap.draw()" and my\_tap.packet().

As mentioned previously, when Wireshark/Tshark dissects a packet, it stores the decoded fields of the current packet in a "protocol tree" structure. To extract information from the tree we must create field extractors. Field extractors are essentially function definitions and are defined via [Field.new\(\)](#). These must be defined outside of the tap.packet() and tap.draw() functions.

To obtain the desired functionality, our script creates and maintains metadata for each connection. This metadata is stored in a table structure we create named “**stream**”. Additionally, we also create a variable to track the number of streams called “**nxt\_stream**”. This variable is initialized to 0, and is incremented each time a new stream is detected. This variable is used for 3 purposes:

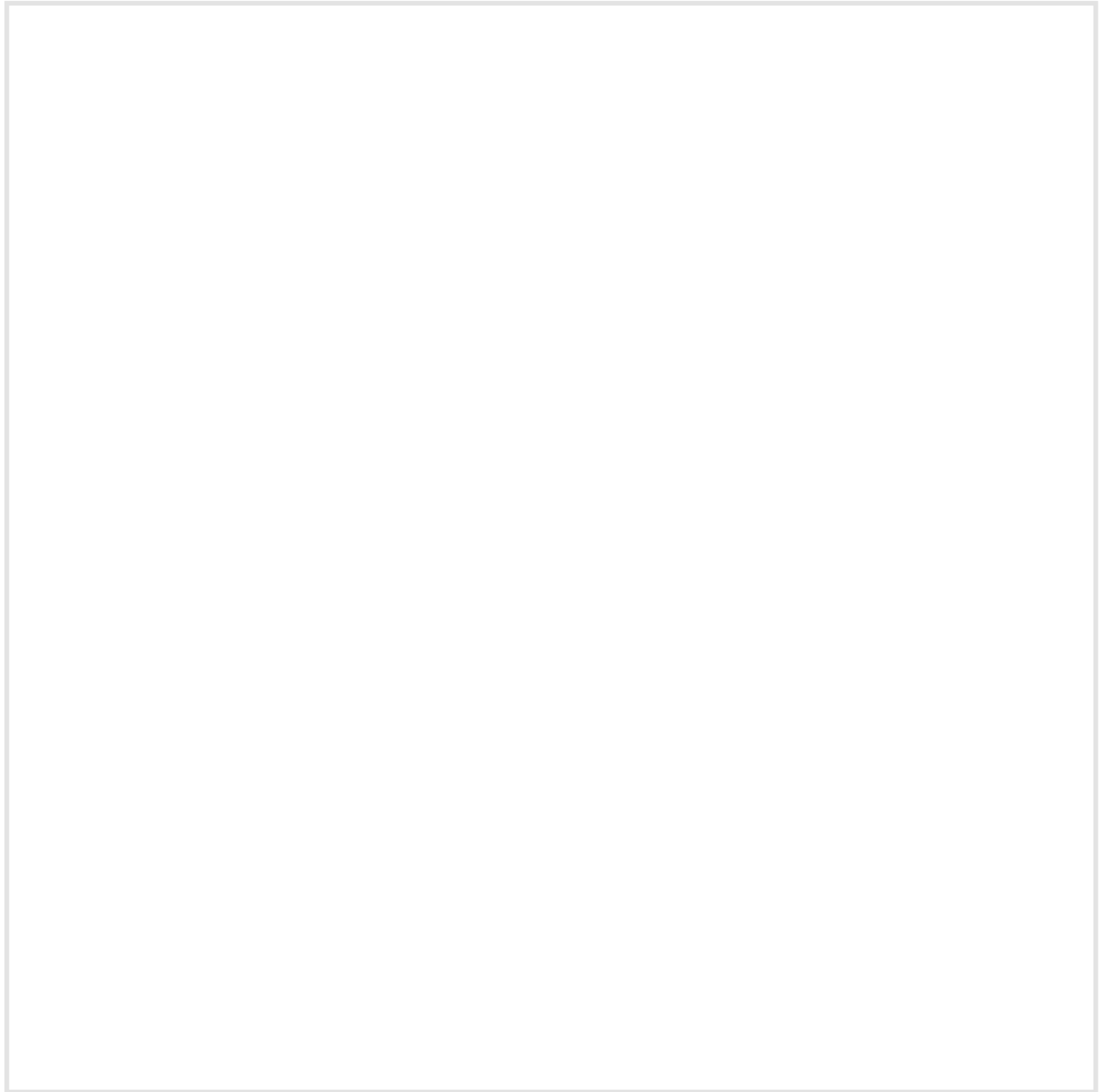
- `tap_packet()` uses `nxt_stream` to determine whether the segment received is part of a new stream. Wireshark/Tshark allocates stream numbers sequentially whenever it sees a segment from a [socket](#) that it has not seen before. As `nxt_stream` contains the value of the next stream we expect to be created, when we receive a segment and it's `tcp.stream` value is lower than `nxt_stream`, we can safely assume that this segment is part of an existing stream and update the stream entry already within our stream table. Otherwise, it's a new stream and we must create a new entry in the table.
- `tap.draw()` uses the value of `nxt_stream` to determine whether any [TCP](#) packets have been seen. If the value of `nxt_stream` = 0, no [TCP](#) traffic has been seen and there is no purpose in creating a column header. We exit `tap.draw()` with a message onscreen indicating that no [TCP](#) segments were not found in the trace.
- `tap.draw()` uses `nxt_stream` to iterate the `stream{}` table. While it is possible to iterate a data structure without knowing it's length using the Lua [pairs\(\)](#) function, Lua does not necessarily store nor retrieve elements in the order in which they were inserted into the table. We set an iterator to 0 (the first numbered stream) and use `nxt_stream` as a limit on a while loop. This ensures that our streams are ordered sequentially.

## tap.packet()

From a high level, Tshark feeds each packet through dissectors, plugins, filters, and then calls our `tap.packet()` function. Thus for each packet, Tshark looks for the function of the listener which we defined named `tap.packet()`. `Tap.packet()` extracts the necessary field data and conditionally updates our `stream{}` table based upon conditional logic. For example, if the segment is out of order, we update the out of order counter.

- If the segment is the first within a stream, we create a “record” for the stream and populate the record accordingly.
- If the packet is part of a stream for which we have an existing record, we update the values of this record accordingly.

The following flow chart outlines the programmatic logic contained within our tap.packet() function.



## Stream data structure

Lua has no concept of lists, structures, etc. Data structures are based entirely on [tables/associative arrays](#) containing key/value pairs. In our example, in order to replicate a record structure we create a table called streams. In the streams table, the key is the stream number and the value is a pointer to the memory location of another table containing the values of a particular stream. Visually, this could be represented as follows:



## tap.draw()

When all packets have been processed, Tshark looks for tap.draw(). Tap.draw reads our stream structure and outputs to the screen accordingly. The following flow chart outlines the programmatic logic contained within our tap.draw() function.



## Additional Notes

All attempts have been made to use local variables to minimize table lookups and unnecessary function calls. If a global variable or function is referenced more than once within tap.draw() or tap.packet(), we assign it to a local variable.

- Within `tap.packet()` and `tap.draw()` we create a local variable to directly reference the record of the stream to which we are working. This eliminates the need to repetitively search (within the stream table) for a specific key to locate the memory location of a given record. *For example, if the frame is retransmission, we need to increment the frame count as well as the retransmission count.* Thus, the creation of a local variable pointing directly to the location of the data related to the current stream eliminates the need to look up the key multiple times in the stream table.
- One interesting line of which I would like call attention, is the following:

```
local conn=tonumber(tostring(tcp_stream()))
```

We are storing the current stream in a local variable. Therefore, we only need to call the field extractor once. However, we need to convert the field extractor output to a string before converting to a number because `tonumber()` expects a string. The return value of `tcpstream()` is userdata and trying to convert it directly results in a nil value. I have spent many hours trying to figure out why my scripts were not working, only to discover that it was due to a variable type mismatch. *For debugging I often add a line such as `"print(type(variable))"` so that I can determine whether my issue is due to a variable type mismatch.*

## Summary

The purpose of writing this blog was to provide a introductory tutorial on Lua tapping. While the example script may seem simplistic, it can easily be extended to provide much more value. Future blogs will seek to extend upon the basic logic presented here. I hope you enjoyed this exercise and look forward to any questions, comments, corrections and/or suggestions.

## 12. Thomas Kager's lua Lesson 2: Filling In the Blanks

### lua Lesson 2 – Filling In The Blanks

Kager

by Thomas P. Kager

In the [first](#) section of this series, I presented a simple Lua script to extract expert data from Tshark. Continuing down this path, a few scripting changes have been made to enhance functionality and introduce new concepts. Specifically:

- White spacing was modified to improve readability and consistency
- Additional extractions have been added to provide contextual information
- A calculated value has been added
- Script timing logic has been added
- Output format has been modified to accommodate for the additional data being presented
- A separate reusable function has been added

#### Download LUA script

After these modifications, our output should resemble the following:



### Spacing and Indentation

Unlike Python, indentation is not a requirement and generally speaking, [LUA only requires spaces between names and keywords](#). Thus, the following code snippets are functionally equivalent:

```
if tcp_analysis_retransmission() ~= nil then l_stream["retr"] = l_stream["retr"] + 1 end
if tcp_analysis_retransmission() ~= nil then
    l_stream["retr"] = l_stream["retr"] + 1
end
```

For the most part, spacing and indentation are a matter of readability and programmer preference.

### Additional Field Extractions

More field extractors have been added to enhance functionality.

- frame.len is now being extracted. This allows us to track the total frame Bytes per connection. Alternately, if we wanted to look at "just" [tcp](#) payload Bytes, we could have chosen [tcp.len](#).
- Each stream entry now tracks the socket information ([ip.src](#), [ip.dst](#), [tcp.srcport](#), [tcp.dstport](#)). This provides greater visual context, than (just) the stream identifier.

### Calculated Value

The total byte calculation and additional fields are logical extensions upon what we did in the 1.0 version of this script. For Bytes, we are adding the current frame length to the current running total stored in bytes and storing this value back into bytes. As you hopefully remember, `l_stream` is a pointer to the current stream record.

```
l_stream["bytes"] = l_stream["bytes"] + tostring(frame_len())
```

### Script Timing

Within `tap.packet()` we introduce a new variable called `start_time`. This variable is initialized to the value returned from the `os.time()` function the first time `tap.packet()` is called. As we call `tap.packet()` (for each frame which contains [TCP](#)) we check to see if this variable exists. If it exists (not nil), we don't overwrite it.

```
if start_time == nil then
    start_time = os.time()
end
```

As mentioned previously, the aforementioned code could have been written on a single line. I chose to extend this over multiple lines to make its functionality more apparent. At the end of `tap.draw()` we subtract `start_time` from the current `os.time()` to arrive at an elapsed time, which we print using `io.write()`.

```
io.write("\n", "Elapsed Time = ", os.time() - start_time, " Seconds", "\n")
```

### Output Formatting

- All `print()` functions have been replaced with `io.write()`. `io.write()` is very similar to `print()`. However, it doesn't place a newline at the end, nor insert tabs between arguments. While the functionality of `print()` can be a convenience (and necessary in certain instances), we are expliciting formatting output and don't want unintended tabs. [This link describes some of the differences between io.write\(\) and print\(\)](#).
- Field lengths were relatively short and predictable in our first script. However, as we introduce fields such as Bytes there is a significant amount of variation in field size which can occur. Thus, we explicitly format the output to ensure that our headers and data align. We use the `string.format()` function to specify field length and justification. [String.format\(\), in conjunction with io.write\(\), behaves similarly to the C printf\(\) function.](#)

### Round() Function

In the first exercise we created two functions: `tap.packet()` and `tap.draw()`. In this exercise we create a new function called `round()`. `Round()` is used to calculate the average frame length per stream. Before discussing `round()`, I want to take a moment to talk about functions. Here are a few simple lines of code that illustrate a simple function and how functions return values for use in other functions.

```
function test_function(your_value)
    new_value = "The value handed to this function was " .. your_value
    return (new_value)
end
io.write("Please enter a value and press return: ")
io.write(test_function(io.read()), "\n")
```

- We initialize our function named `test_function`. Functions (like other code blocks) are delimited by end statements.
- We print a line telling the user what to do. *"Please enter a value and press return: "*
- The last line is a bit more complex. We need to work from the innermost function outwards.
  - `io.read()` waits for user input (delimited by a return key). Once the return sequence is detected, the value is passed as an argument to `test_function()`.
  - `test_function()` accepts this value and stores it in "your\_value", appends it to a string and return(s) it. *The "..." (two dots) is the Lua concatenation operator.*
  - Lastly, `io.write` displays this value onscreen. *Function arguments are separated by commas. The "\n" is the newline.*

Let's talk about `round()`. LUA has no built-in mathematical rounding function. To overcome this limitation, we create our own rounding function using the modulus operator and the `math.ceil()` and `math.floor()` functions. *Modulus "%" is the remainder from a division operation.* Thus, if we divide a number by 1 we will end up with the decimal remainder. If this remainder is greater than or equal to .5, we round to the nearest integer. Conversely, less than .5, we round down.

```
function round
    if n % 1 >= 0.5 then return math.ceil else return math.floor end
end
```

While we could place the rounding code within `tap.draw()`, we may use this function again as this script evolves. I am not sure yet.

## One last note

Within `tap.draw()`, there is section of code within one of the `io.write()` calls which may appear a bit unusual. The full line is a bit long, so I am just going to concentrate on the snippet of interest. Specifically, I am focusing in on the section of code involving the `round()` function. To get the frame size average we would expect to divide the total bytes by total frames. This would, in turn, be passed to our `round()` function which would round it off to the nearest integer. Thus, we might anticipate the following:

```
round(_stream["bytes"] / _stream["frames"])
```

However, the code is written a bit differently:

```
round(_stream["bytes"] * (1 / _stream["frames"]))
```

There is a reason for this, has to do with the fact that Lua is much faster at multiplying than dividing. Thus, it's a performance optimization [trick](#).

## Summary

That's it for now. We will continue to extend our script in future installments. I hope you found this interesting and useful. As always, feedback is appreciated.